Stony Brook University

ESE 326: Algorithmic Electronic Design

Final Project

Group members: John Abraham, Tianqin Fu

Date: 05/04/2024

Introduction:

Graph partitioning is a fundamental problem in computer science with applications spanning various domains such as VLSI design, scientific computing, and network optimization. Given a graph, the goal of partitioning is to divide its vertices into disjoint subsets while minimizing certain objectives, such as minimizing the number of edges cut between partitions or ensuring load balance across partitions.

The Fiduccia-Mattheyses algorithm, often referred to as FM algorithm, is a heuristic algorithm used for partitioning in combinatorial optimization. It's commonly applied in electronic design automation for circuit partitioning and optimization. The FM algorithm is best known for its simplicity and efficiency. Developed in the context of VLSI circuit design, the FM algorithm iteratively refines an initial partition by greedily moving vertices between partitions based on a gain function until no further improvement can be achieved. While it doesn't guarantee finding the optimal solution, it often produces high-quality partitions in a reasonable amount of time, making it a popular choice for practical applications where exact solutions are computationally infeasible.

Objective:

To implement the Fiduccia-Mattheyses partitioning algorithm, minimize the size of the cutset in gate-level designs while meeting area constraints assigned to the partitions.

<u>Flowcharts</u>:

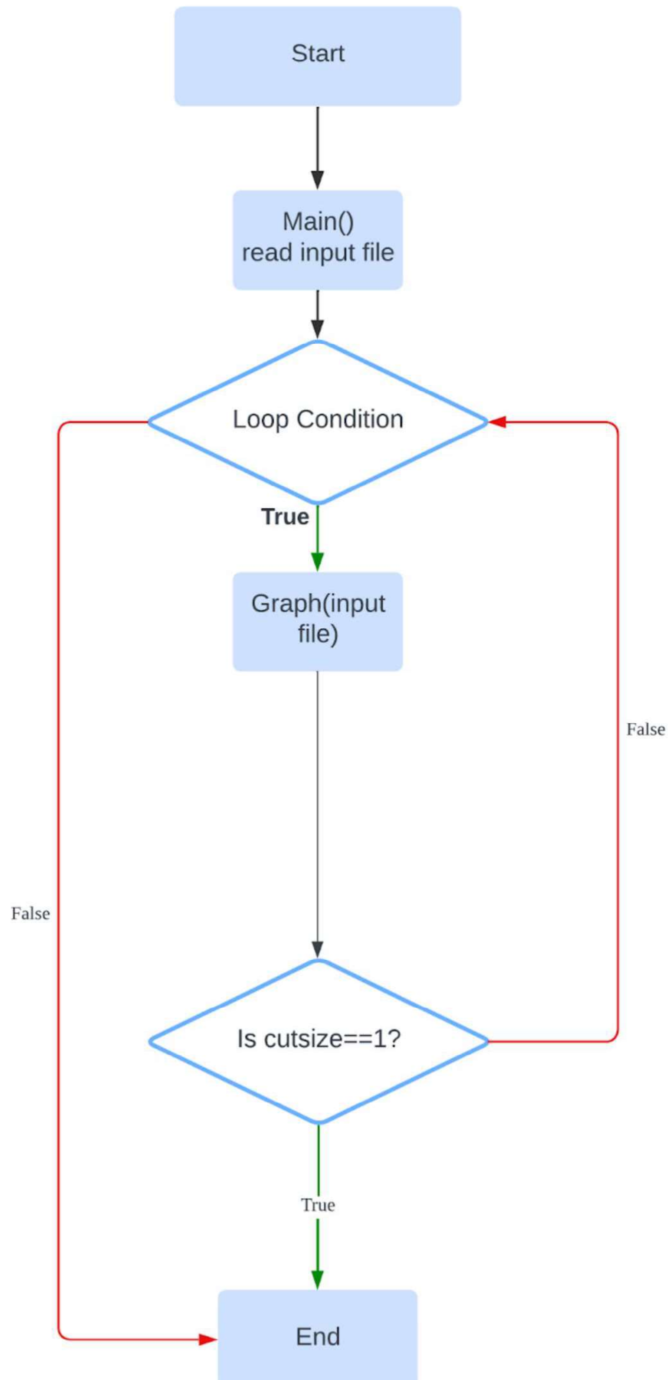


Fig.1: General flow of the algorithm

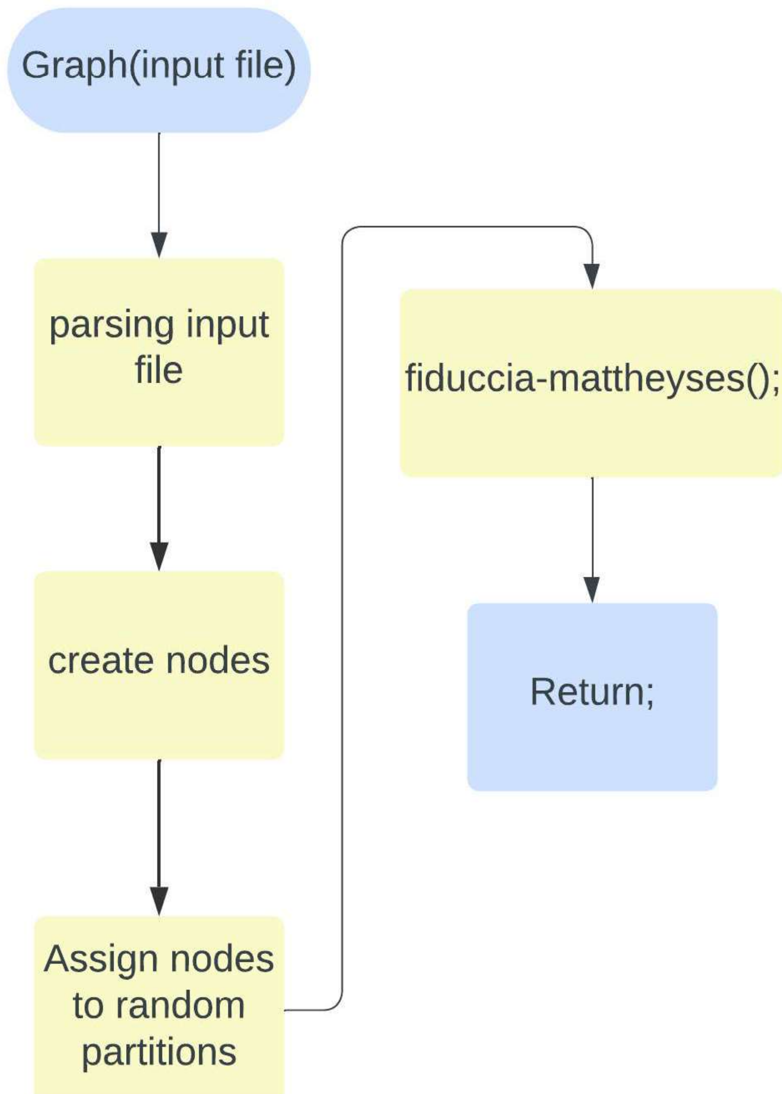


Fig.2: Flow of Graph() function

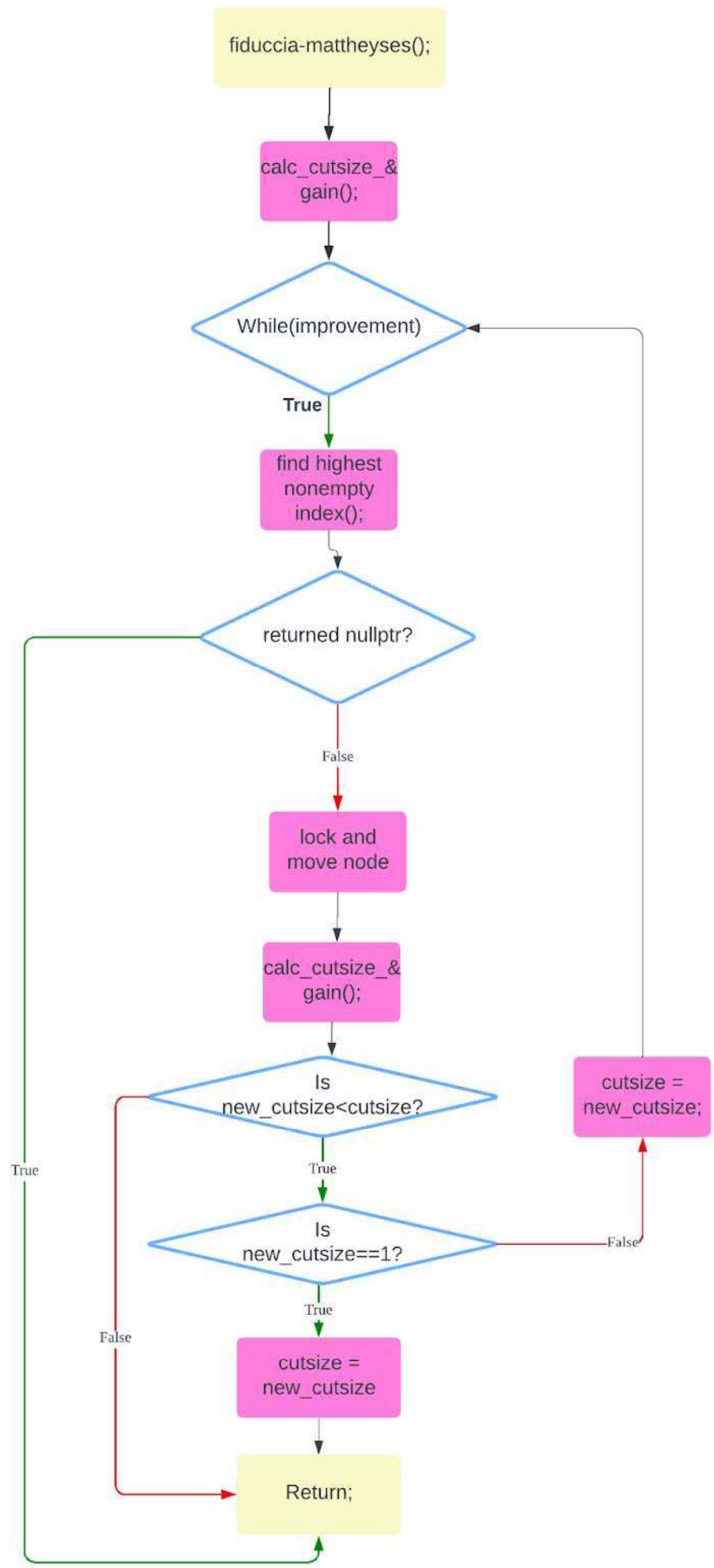
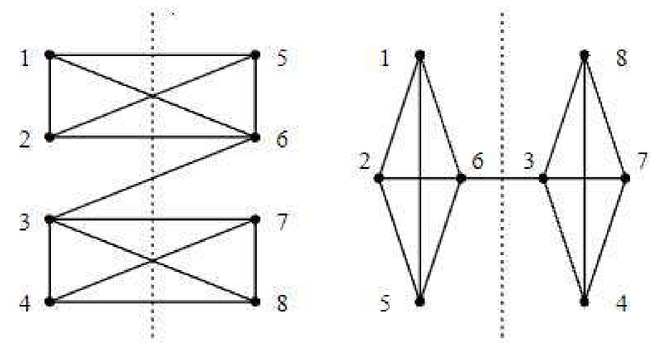


Fig.3: Flow of the Fiduccia-Mattheyses algorithm

Explanation of Input File:
Input File 1:

```
8
7
0 1 5 4
1 4 5
4 5
5 2
2 6 3 7
3 6 7
6 7
```



(a) Initial Bisections (b) Final Bisections

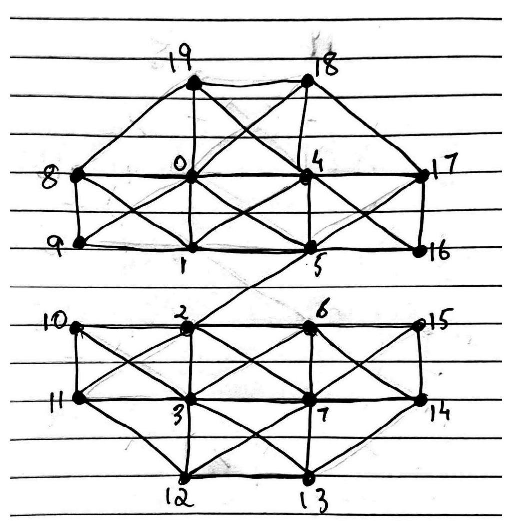
We had issues understanding the ISPD98 Circuit Benchmark Suite and therefore to tackle this minor issue, we decided to create our own input file format for a netlist. We used the example from the textbook where we know the minimum cut size possible is 1. By doing so, we were able to keep track of whether our algorithm was running correctly or not. The example in the textbook starts numbering the nodes from 1, but we started numbering from 0 to make things simpler for coding.

The first line in the input file gives us the number of vertices. The second line gives you the number of nets in the circuit. Each line after that is a net where the first number is the source node of the net and the numbers after that are the destination nodes from the source node.

For example, in the first net, 0 is the source node, and its connections are 0→1, 0→5, and 0→4.

Input File 2: An extended version of input file 1, where we know the minimum cutset size is 1.

```
20
19
0 1 5 4
1 4 5
4 5
5 2
2 6 3 7
3 6 7
6 7
8 0 1 9 18
9 0 1
10 2 3 11
11 2 3 12
12 3 7 13
13 3 7 14
14 7 6 15
15 7 6
16 5 4 17
17 5 4 18
18 4 0 19
19 4 0
```



<u>Sample Output using textbook example:</u>

The ‘r’ values are the random values used to assign nodes to partitions. When we display the partitions, the first number is the node and the second is the boolean value of False or True (0 or 1). Partition 1 contains all the nodes that were assigned False and partition 2 contains all the nodes that were assigned True. We then visually check if the nodes in the circuit are correctly assigned to their partitions by printing the node and its respective partition.

```
**********************ITERATION 1*******************************
r:0.351508
r:0.505971
r:0.687558
r:0.342601
r:0.429402
r:0.47279
r:0.197051
r:0.765581
pmax: 4
partition1:
00
30
40
50
60
partition2:
11
21
71
in the circuit:
00
11
21
30
40
50
60
71
```

```
node:0 gain:-1 locked: 0 partition:0
node:1 gain:3 locked: 0 partition:1
node:2 gain:2 locked: 0 partition:1
node:3 gain:1 locked: 0 partition:0
node:4 gain:-1 locked: 0 partition:0
node:5 gain:0 locked: 0 partition:0
node:6 gain:1 locked: 0 partition:0
node:7 gain:1 locked: 0 partition:1
cutsize: 8
--------------------------------------------------
moving node...
node moved:1
node:0 gain:-3 locked: 0 partition:0
node:1 gain:-3 locked: 1 partition:0
node:2 gain:2 locked: 0 partition:1
node:3 gain:1 locked: 0 partition:0
node:4 gain:-3 locked: 0 partition:0
node:5 gain:-2 locked: 0 partition:0
node:6 gain:1 locked: 0 partition:0
node:7 gain:1 locked: 0 partition:1
cutsize: 5
--------------------------------------------------
moving node...
node moved:2
node:0 gain:-3 locked: 0 partition:0
node:1 gain:-3 locked: 1 partition:0
node:2 gain:-2 locked: 1 partition:0
node:3 gain:-1 locked: 0 partition:0
node:4 gain:-3 locked: 0 partition:0
node:5 gain:-4 locked: 0 partition:0
node:6 gain:-1 locked: 0 partition:0
node:7 gain:3 locked: 0 partition:1
cutsize: 3
--------------------------------------------------
```

```
moving node...
node moved:7
node:0 gain:-3 locked: 0 partition:0
node:1 gain:-3 locked: 1 partition:0
node:2 gain:-4 locked: 1 partition:0
node:3 gain:-3 locked: 0 partition:0
node:4 gain:-3 locked: 0 partition:0
node:5 gain:-4 locked: 0 partition:0
node:6 gain:-3 locked: 0 partition:0
node:7 gain:-3 locked: 1 partition:0
cutsize: 0
--------------------------------------------------
moving node...
node moved:0
node:0 gain:3 locked: 1 partition:1
node:1 gain:-1 locked: 1 partition:0
node:2 gain:-4 locked: 1 partition:0
node:3 gain:-3 locked: 0 partition:0
node:4 gain:-1 locked: 0 partition:0
node:5 gain:-2 locked: 0 partition:0
node:6 gain:-3 locked: 0 partition:0
node:7 gain:-3 locked: 1 partition:0
cutsize: 3
no more improvements.
Final cutsize after Fiduccia-Mattheyses: 3

**********************ITERATION 2*******************************
r:0.623804
r:0.479471
r:0.81834
r:0.646885
r:0.520252
r:0.152983
r:0.768068
```

```
r:0.698782
pmax: 4
partition1:
10
50
partition2:
01
21
31
41
61
71
in the circuit:
01
10
21
31
41
50
61
71
```

```
node:0 gain:1 locked: 0 partition:1
node:1 gain:1 locked: 0 partition:0
node:2 gain:-2 locked: 0 partition:1
node:3 gain:-3 locked: 0 partition:1
node:4 gain:1 locked: 0 partition:1
node:5 gain:2 locked: 0 partition:0
node:6 gain:-3 locked: 0 partition:1
node:7 gain:-3 locked: 0 partition:1
cutsize: 5
------------------------------------------------
moving node...
node moved:5
node:0 gain:-1 locked: 0 partition:1
node:1 gain:3 locked: 0 partition:0
node:2 gain:-4 locked: 0 partition:1
node:3 gain:-3 locked: 0 partition:1
node:4 gain:-1 locked: 0 partition:1
node:5 gain:-2 locked: 1 partition:1
node:6 gain:-3 locked: 0 partition:1
node:7 gain:-3 locked: 0 partition:1
cutsize: 3
------------------------------------------------
moving node...
node moved:1
node:0 gain:-3 locked: 0 partition:1
node:1 gain:-3 locked: 1 partition:1
node:2 gain:-4 locked: 0 partition:1
node:3 gain:-3 locked: 0 partition:1
node:4 gain:-3 locked: 0 partition:1
node:5 gain:-4 locked: 1 partition:1
node:6 gain:-3 locked: 0 partition:1
node:7 gain:-3 locked: 0 partition:1
cutsize: 0
------------------------------------------------
```

```
moving node...
node moved:0
node:0 gain:3 locked: 1 partition:0
node:1 gain:-1 locked: 1 partition:1
node:2 gain:-4 locked: 0 partition:1
node:3 gain:-3 locked: 0 partition:1
node:4 gain:-1 locked: 0 partition:1
node:5 gain:-2 locked: 1 partition:1
node:6 gain:-3 locked: 0 partition:1
node:7 gain:-3 locked: 0 partition:1
cutsize: 3
no more improvements.
Final cutsize after Fiduccia-Mattheyses: 3

**********************ITERATION 3*********************************
r:0.432698
r:0.825707
r:0.871943
r:0.64379
r:0.758719
r:0.650321
r:0.385127
r:0.0753493
pmax: 4
partition1:
00
60
70
partition2:
11
21
31
41
51
```

```
in the circuit:
00
11
21
31
41
51
60
70
node:0 gain:3 locked: 0 partition:0
node:1 gain:-1 locked: 0 partition:1
node:2 gain:0 locked: 0 partition:1
node:3 gain:1 locked: 0 partition:1
node:4 gain:-1 locked: 0 partition:1
node:5 gain:-2 locked: 0 partition:1
node:6 gain:1 locked: 0 partition:0
node:7 gain:1 locked: 0 partition:0
cutsize: 7
-------------------------------------------------
moving node...
node moved:0
node:0 gain:-3 locked: 1 partition:1
node:1 gain:-3 locked: 0 partition:1
node:2 gain:0 locked: 0 partition:1
node:3 gain:1 locked: 0 partition:1
node:4 gain:-3 locked: 0 partition:1
node:5 gain:-4 locked: 0 partition:1
node:6 gain:1 locked: 0 partition:0
node:7 gain:1 locked: 0 partition:0
cutsize: 4
-------------------------------------------------
```

```
moving node...
node moved:3
node:0 gain:-3 locked: 1 partition:1
node:1 gain:-3 locked: 0 partition:1
node:2 gain:2 locked: 0 partition:1
node:3 gain:-1 locked: 1 partition:0
node:4 gain:-3 locked: 0 partition:1
node:5 gain:-4 locked: 0 partition:1
node:6 gain:-1 locked: 0 partition:0
node:7 gain:-1 locked: 0 partition:0
cutsize: 3
-------------------------------------------------
moving node...
node moved:2
node:0 gain:-3 locked: 1 partition:1
node:1 gain:-3 locked: 0 partition:1
node:2 gain:-2 locked: 1 partition:0
node:3 gain:-3 locked: 1 partition:0
node:4 gain:-3 locked: 0 partition:1
node:5 gain:-2 locked: 0 partition:1
node:6 gain:-3 locked: 0 partition:0
node:7 gain:-3 locked: 0 partition:0
cutsize: 1
minimum partitioning found.
Final cutsize after Fiduccia-Mattheyses: 1
Minimum cutsize found.

C:\Users\jobif\OneDrive\Documents\John College\326\fm_version2\
To automatically close the console when debugging stops, enable
ging stops.
Press any key to close this window . . .
```

This output took 3 iterations of Fiduccia-Mattheyses to obtain the minimum cutset size of 1. We ran the code at different instances of time and the highest number of iterations we saw it took to reach the minimum cutset size of 1 was 7. The number of iterations it took heavily depended on the initial random partitions. If the random partitions are good, the algorithm could find the minimum cutset size in the first iteration itself.

As our input was a small net, the number of iterations our algorithm took to find the minimum cutset size was always less than 10. When the net increases in size, the number of iterations it will take to find the minimum cutset size will also increase.

Output for Input File 2:

This output found the minimum cutset size within 2 iterations of Fiduccia Mattheyses.
The maximum number of iterations ever taken for the second input file was 12, which is not bad for an input file that is more than double the size of the first input file. This also shows us that our algorithm is not increasing exponentially with size. However, the fact still remains that the number of iterations of performing Fidducia-Matheyses is heavily dependent on the initial random partition. Below is a pic of the last node move of the second iteration that found the minimum cutset size of 1.

```
--------------------------------------------------
moving node...
node moved:15
node:0 gain:-7 locked: 0 partition:0
node:1 gain:-5 locked: 0 partition:0
node:2 gain:-4 locked: 1 partition:1
node:3 gain:-7 locked: 0 partition:1
node:4 gain:-7 locked: 0 partition:0
node:5 gain:-4 locked: 1 partition:0
node:6 gain:-5 locked: 0 partition:1
node:7 gain:-7 locked: 0 partition:1
node:8 gain:-4 locked: 1 partition:0
node:9 gain:-3 locked: 0 partition:0
node:10 gain:-3 locked: 0 partition:1
node:11 gain:-4 locked: 0 partition:1
node:12 gain:-4 locked: 1 partition:1
node:13 gain:-4 locked: 0 partition:1
node:14 gain:-4 locked: 1 partition:1
node:15 gain:-3 locked: 1 partition:1
node:16 gain:-3 locked: 0 partition:0
node:17 gain:-4 locked: 1 partition:0
node:18 gain:-5 locked: 1 partition:0
node:19 gain:-3 locked: 0 partition:0
cutsize: 1
minimum partitioning found.
Final cutsize after Fiduccia-Mattheyses: 1
Minimum cutsize found.

Time taken: 180.339 milliseconds

C:\Users\jobif\OneDrive\Documents\John College\326\fm_version2\x64
To automatically close the console when debugging stops, enable To
ging stops.
Press any key to close this window . . .
```

Possible Implementation Issues:

1. Code Organization:
   - Break down complex functions into smaller, more modular functions. This would not only improve readability but also allow for easier optimization of individual components.
   - Eliminate redundant code and calculations. Look for opportunities to reuse results or streamline calculations to avoid unnecessary overhead.

2. Loop Consolidation:
   - Where possible, consider consolidating multiple loops into one to reduce overhead. For example, when we iterate over the same data multiple times for different purposes, see if it is possible to combine those operations into a single loop.

Conclusion:

This implementation of the Fiduccia-Mattheyses partitioning algorithm for gate-level designs has completed the designed intention of partitioning a netlist into 2 partitions. In testing multiple times, it was observed that the cutset size decreased from a max cutset size of 9 to 1. The result suggests that this algorithm is functional.

The following can be done to improve the result of this implementation furthermore:

1. Initial Partitioning Strategy: The FM algorithm often starts with an initial partition, which can significantly affect its performance. Developing better strategies for generating initial partitions, such as using more sophisticated heuristics or incorporating problem-specific knowledge, could improve the overall effectiveness of the algorithm.
2. Parameter tuning: By fine tuning the parameters such as ratio of areas or numbers of iterations, The algorithm may produce a more desirable result.
3. Parallelization: Graph partitioning is inherently parallelizable, and exploiting parallelism can lead to significant speedups, especially for large graphs. Investigating parallel versions of the FM algorithm or integrating it with parallel computing frameworks could improve its scalability and performance on modern multicore and distributed systems.
4. Balancing Constraints: Ensuring that the partitions produced by the algorithm are balanced in terms of size and/or other constraints is crucial, especially in applications like VLSI design where balance is essential for performance. Developing techniques to enforce balance constraints more effectively without sacrificing partition quality is an area for improvement.

Bibliography:

Alpert, Charles J. The ISPD98 Circuit Benchmark Suite, vlsicad.ucsd.edu/UCLAWeb/cheese/ispd98.html.

N. Sherwani, "Algorithms for VLSI Physical Design Automation", Kluwer, 1999.

Appendix:

<u>An overview of all the functions:</u>

```
class Graph {
private:
    int numVertices;    //number of vertices
    int cutsize;
    int pmax=0;         //maximum gain possible.
    int offset = 0; //an offset is required to store both +ve and -ve gains.
    vector<node> nodes;

    vector<node> partition1;
    vector<node> partition2;
    circuit circuit_nets;   //each index of circuit_nets holds a net.
    Bucket gainBucket;      //the Bucket structure.


public:
    Tabnine
    int calc_Cutsize_and_gain(){ ... }

    // Constructor to read the number of vertices from a file and initialize everything.
    Graph(const string& filename){ ... }


    /*Function to perform the Fiduccia - Mattheyses algorithm.
    * First, it selects the highest gain non empty node.
    * then it locks the node and moves it to the opposite partition.
    * then it recalculates the new gains and cutsize for all the nodes.
    * If the new cutsize is less than the current cutsize then
    * it saves the new cutsize and performs another node move.
    */
    Tabnine
    void fiduccia_mattheyses(){ ... }

    /*This function is used to check if all the nodes
    * in a given index of our gainBucket are locked or not.
    * It returns a pair of values that are used by the
    * findHighestNonEmptyIndex() function.
    * The first return value in the pair returns true if
    * all the nodes are locked and false otherwise.
    * The second value returns the node that is free or
    * returns nullptr if no free node is found.
    */
    Tabnine
    pair<bool, node*> check_all_nodes_locked(int i){ ... }

    /*This function is used to find the highest index where
    * a non-empty node can be found.
    */
    node& findHighestNonEmptyIndex(){ ... }

    Tabnine
    int return_cutsize() {
        return cutsize;
    }

};
```

Structs.h:

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <unordered_set>
using namespace std;

//NOTE: When I use "", I am referring to the exact name of the variable I used in the code.

struct node {
    bool partition_num;
    int value;
    bool locked = false;
};

struct Net {
/*A net has a source node and may have multiple destination nodes.
The destination nodes are stored in vector<node> tv  variable.
The source node (eg:0) can be also be destination node for some other node (eg:7).
The unordered_set stores all the nodes in "tv" along with nodes like "7".
We use the "tv" variable to calculate the cutsize.
We use the "unordered_set" to calculate the gain for each net's source node.
You can think of "tv" as a subset of "connected" where all the nodes in "tv"
are present in "connected" but not vice versa.
    */
    node source;
    vector<node> tv;    //to_vertices
    unordered_set<int> connected;
};

/*A bucket is a 2d vector of pointers to nodes.
the first index would be the gain values ranging from +pmax to -pmax.
At each gain index, we will have a vector of pointers
pointing to the nodes that have that gain value.
*/
class Bucket {
public:
    Bucket() {};
    vector<vector<node*>> b;
};

class circuit
{
public:
    circuit() {};

```

```
    //resize the length of circuit_nets.
    void resizeNet(int size) {
        circuit_nets.resize(size);
    }

    // Define operator[] to access and modify elements at specific indices of circuit_nets.
    Net& operator[] (size_t index) {
        if (index >= circuit_nets.size()) {
            cout << "ERROR:out of range of circuit_nets" << endl;
            exit(1);
        }
        return circuit_nets[index];
    }

    // Define begin() and end() functions to provide iterator interface
    vector<Net>::iterator begin() {
        return circuit_nets.begin();
    }

    vector<Net>::iterator end() {
        return circuit_nets.end();
    }
private:
    vector<Net> circuit_nets;
};

```

FM source file:

```cpp
#include "structs.h"
#include <iostream>
#include <fstream>
#include <vector>
#include <sstream> // for stringstream
#include <cstdlib> // for rand() and srand()
#include <ctime>   // for time()
#include <algorithm>
#include <unordered_set>
#include <random>
#include <chrono>
using namespace std;



// Create a random number engine and seed it
std::random_device rd;
std::mt19937 gen(rd());
// Create a uniform distribution for floating-point numbers between 0 and 1 for randomizing the partition.
std::uniform_real_distribution<double> dis(0.0, 1.0);


//NOTE: When I use "", I am referring to the exact name of the variable I used in the code.


class Graph {
private:
    int numVertices;    //number of vertices
    int cutsize;
    int pmax=0;         //maximum gain possible.
    int offset = 0; //an offset is required to store both +ve and -ve gains.
    vector<node> nodes;

    vector<node> partition1;
    vector<node> partition2;
    circuit circuit_nets;   //each index of circuit_nets holds a net.
    Bucket gainBucket;      //the Bucket structure.


public:
    int calc_Cutsize_and_gain() {
        /*we need to reset the gainBucket so that the previous gain values
        for nodes are removed.
        we use a temp Bucket to replace the current Bucket.
        THe outer loop iterates through every net.
        In each net, the first inner loop iterates through all the nodes
```

```
        that are connected to the source node.
        the second inner loop iterates through all the nodes that are
        connected to the current source node and checks whether they are
        in the same partition or not.
        */

        Bucket temp;
        temp.b.resize(2 * pmax + 1);

        int cutsize = 0;
        for (auto& net : circuit_nets) {
            int out_edge = 0;
            int in_edge = 0;
            bool partition = net.source.partition_num;
            for (auto& x : net.connected) {
                if (partition != circuit_nets[x].source.partition_num) {
                    out_edge++;
                }
                else {
                    in_edge++;
                }
            }
            for (auto& x : net.tv) {
                if (partition != circuit_nets[x.value].source.partition_num) {
                    cutsize++;
                }
            }
            cout << "node:" << net.source.value<<" gain:"<< out_edge - in_edge << " locked: " << net.source.locked <<" partition:" << net.source.partition_num << endl;
    //You can read the next instruction as:
    // Calculate the gain, add an offset to it for the index of the Bucket,
    // take that index of the bucket,
    // push back a pointer to the source node of the current net.
    // We sent pointer because the bucket holds pointers to the nodes.
            temp.b[(out_edge - in_edge) + offset].push_back(&(net.source)); //the offset is necessary for -ve gains.

        }
        gainBucket = temp;
        cout << "cutsize: " << cutsize << endl;
        return cutsize;
    }

    // Constructor to read the number of vertices from a file and initialize everything.
    Graph(const string& filename) {
```

```cpp
        ifstream inFile(filename);
        if (!inFile.is_open()) {
            cerr << "Error opening file " << filename << endl;
            exit(1);
        }

        // Read number of vertices
        inFile >> numVertices;

        // Initialize nodes vector
        nodes.resize(numVertices);

        int numNets;
        inFile >> numNets;
        circuit_nets.resizeNet(numVertices);

        for (int i = 0; i < numVertices; ++i) {
            nodes[i].value = i;
            nodes[i].partition_num = false; //assign all nodes to partition 1 initially.
            circuit_nets[i].source.value = i;
        }




        // Read nets
        string line;
        int source;
        node source_node;
        bool p = false;  //assign all nodes to partition 1 initially.

        getline(inFile, line);
        for (int i = 0; i < numNets; ++i) {
            getline(inFile, line);
            stringstream ss(line);

            ss >> source;
            source_node.value = source;
            source_node.partition_num = p;

            unordered_set<int> temp;
            vector<node> tv;
            node n;
            int node_value;



            while (ss >> node_value) {
```

```
                n.value = node_value; //assigns node value
                n.partition_num = p;  //assigns to partition 1.
                tv.push_back(n);  //add to vector of destination nodes
                temp.insert(node_value); //all the nodes in "tv" must be in "connected".
            }
            Net N1;
            N1.source = source_node;
            N1.tv = tv;
            N1.connected = temp;

            if (tv.size() > pmax) {
                pmax = tv.size();   //for bucket.
            }
            circuit_nets[source] = N1;

        }

        inFile.close();

        //create random partitions.
        for (auto& x : circuit_nets) {
            double r = dis(gen);
            cout << "r:" << r << endl;
            if (r < 0.5) {
                p = false;
            }
            else { p = true; }

            x.source.partition_num = p;
            if (x.source.partition_num == false) {
                partition1.push_back(x.source);
            }
            else {
                partition2.push_back(x.source);
            }

        /*This is where we initialize the unordered_set "connected" for
        calculating the gain later. In each net of circuit_nets, we
        iterate through the destination nodes(eg: y) of the net and add
        the current source node(i.e., x.source.value) to the unordered_set

        of the current destination node, i.e., y.
        We then check for the largest connected node to set as pmax.
        */
            for (auto& y : x.tv) {
                circuit_nets[y.value].connected.insert(x.source.value);
                if (circuit_nets[y.value].connected.size() > pmax) {
                    pmax = circuit_nets[y.value].connected.size();
```

```
                }
            }
        }

        /*the offset is equal to pmax because we want the gainBucket to hold
        values from +pmax to -pmax.*/
        offset = pmax;
        cout << "pmax: " << pmax << endl;
        gainBucket.b.resize(2 * pmax + 1);


/*This is where we actually assign the nodes to partitions
* The indices of circuit_nets correspond to the respective source nodes
* For example, circuit_nets[1] holds the net N whose source node is 1.
*/
        cout << "partition1:" << endl;
        for (auto& x : partition1) {
            circuit_nets[x.value].source.partition_num = x.partition_num;
            cout << circuit_nets[x.value].source.value << circuit_nets[x.value].source.partition_num << endl;
        }
        cout << "partition2:" << endl;
        for (auto& x : partition2) {
            circuit_nets[x.value].source.partition_num = x.partition_num;
            cout << circuit_nets[x.value].source.value << circuit_nets[x.value].source.partition_num << endl;
        }
        cout << "in the circuit:" << endl;
        for (auto& x : circuit_nets) {
            cout << x.source.value << x.source.partition_num << endl;
        }


        // Perform Fiduccia-Mattheyses algorithm
        fiduccia_mattheyses();

        //cutsize = calc_Cutsize_and_gain();

        // Output final cutsize
        cout << "Final cutsize after Fiduccia-Mattheyses: " << cutsize << endl;
    }


    /*Function to perform the Fiduccia - Mattheyses algorithm.
    * First, it selects the highest gain non empty node.
    * then it locks the node and moves it to the opposite partition.
    * then it recalculates the new gains and cutsize for all the nodes.
```

```cpp
* If the new cutsize is less than the current cutsize then
* it saves the new cutsize and performs another node move.
*/
void fiduccia_mattheyses() {

    // Calculate initial cutsize and gain
    cutsize=calc_Cutsize_and_gain();

    bool improvement = true;

    while(improvement){
            cout << "-------------------------------------------------" << endl;
            cout << "moving node..." << endl;

            node& k = findHighestNonEmptyIndex();
            if (&k == nullptr) {
                cout << "all nodes locked." << endl;
                return;
            }
            cout << "node moved:" << k.value << endl;
            k.locked = true;
            k.partition_num = !k.partition_num; //move to other partition.
            //recalculate new gains for all nodes
            int new_cutsize = calc_Cutsize_and_gain();
            if (new_cutsize < cutsize) {
                if (new_cutsize == 1) {
                    cutsize = new_cutsize;
                    cout << "minimum partitioning found." << endl;
                    return;
                }
                if (new_cutsize > 1) { //if cutsize is 0, dont modify the current cutsize because there might free nodes still present.
                    cutsize = new_cutsize;
                }
            }
            else {
                cout << "no more improvements." << endl;
                improvement = false;
            }
    }
}

/*This function is used to check if all the nodes
* in a given index of our gainBucket are locked or not.
* It returns a pair of values that are used by the
```

```cpp
    * findHighestNonEmptyIndex() function.
    * The first return value in the pair returns true if
    * all the nodes are locked and false otherwise.
    * The second value returns the node that is free or
    * returns nullptr if no free node is found.
    */
    pair<bool, node*> check_all_nodes_locked(int i) {
        for (auto& x_ptr : gainBucket.b[i]) {
            node* x = x_ptr;
            if (!x->locked) { // Check if node is not locked
                return make_pair(false, x); // Return false and node pointer if found
            }
        }
        return make_pair(true, nullptr); // Return true and nullptr if all nodes are locked
    }

    /*This function is used to find the highest index where
    * a non-empty node can be found.
    */
    node& findHighestNonEmptyIndex() {
        int highestIndex = 0;
        node* k=nullptr;
        bool found = false;
        int i = 0;
        while(i < gainBucket.b.size()){ //iterate through every gain index.
            if (!gainBucket.b[i].empty()) {      // if the bucket index is not empty
                pair<bool, node*> result = check_all_nodes_locked(i);      //check for free nodes within that gain index.
                if (!result.first){         //if free node found
                    highestIndex = i;       //highest non empty index
                    k = result.second;
                }
            }
            ++i;
        }
        return *k;
    }

    int return_cutsize() {
        return cutsize;
    }

};

//Number of iterations to run Fidducia-Matheyses.
```

```
const int NUM_OF_ITERATIONS = 10;
int main() {
    //Start timer.
    auto start = std::chrono::high_resolution_clock::now();

    string file = "input_net.txt";
    int final_cutsize = INT_MAX;
    for (int i = 1; i <= NUM_OF_ITERATIONS; i++) {
        cout << endl;
        cout << "***********************ITERATION " << i <<
          "********************************" << endl;
        Graph g(file);
        if (g.return_cutsize() < final_cutsize) {
            final_cutsize = g.return_cutsize();
        }
        if (g.return_cutsize() == 1) {
            cout << "Minimum cutsize found." << endl;
            break;
        }
    }

    // Stop timer
    auto end = std::chrono::high_resolution_clock::now();
    // Calculate the duration
    std::chrono::duration<double> duration = end - start;
    // Convert duration to milliseconds
    double milliseconds = duration.count() * 1000.0;
    // Output the time taken
    cout << endl;
    std::cout << "Time taken: " << milliseconds << " milliseconds" <<
      std::endl;
}
```